# Transparent Dynamic Optimization

Vasanth Bala, Evelyn Duesterwald, Sanjeev Banerjia
HP Laboratories Cambridge
HPL-1999-77
June, 1999

E-mail: vas@hpl.hp.com

dynamic optimization, compilers, code cache, runtime optimization, performance

Dynamic optimization refers to the *runtime* optimization of a *native* program binary. This paper describes the design and implementation of Dynamo, a prototype dynamic optimizer that is capable of optimizing a native program binary at runtime. Dynamo is a realistic implementation, not a simulation, that is written entirely in user-level software, and runs on a PA-RISC machine under the HPUX operating system.

Dynamo does not depend on any special programming language, compiler, operating system or hardware support. The program binary is not instrumented and is left untouched during Dynamo's operation. Dynamo observes the program's behavior through interpretation to dynamically select hot instruction traces from the running program. The hot traces are optimized using low-overhead optimization techniques and emitted into a software code cache. Subsequent instances of these traces cause the cached version to be executed, resulting in a performance boost.

Contrary to intuition, we demonstrate that it is possible to use a piece of software to improve the performance of a native, statically optimized program binary, *while it is executing*. Dynamo not only speeds up real application programs, its performance improvement is often quite significant. For example, the performance of many +O2 optimized SPECint95 binaries running under Dynamo is comparable to the performance of their +O4 optimized version running without Dynamo.

# Transparent Dynamic Optimization

Vasanth Bala
Evelyn Duesterwald
Sanjeev Banerjia

Hewlett-Packard Labs
Cambridge, MA 02142

### Abstract

Dynamic optimization refers to the *runtime* optimization of a *native* program binary. This paper describes the design and implementation of Dynamo, a prototype dynamic optimizer that is capable of optimizing a native program binary at runtime. Dynamo is a realistic implementation, not a simulation, that is written entirely in user-level software, and runs on a PA-RISC machine under the HPUX operating system.

Dynamo does not depend on any special programming language, compiler, operating system or hardware support. The program binary is not instrumented and is left untouched during Dynamo's operation. Dynamo observes the program's behavior through interpretation to dynamically select hot instruction traces from the running program. The hot traces are optimized using low-overhead optimization techniques and emitted into a software code cache. Subsequent instances of these traces cause the cached version to be executed, resulting in a performance boost.

Contrary to intuition, we demonstrate that it is possible to use a piece of software to improve the performance of a native, statically optimized program binary, *while it is executing*. Dynamo not only speeds up real application programs, its performance improvement is often quite significant. For example, the performance of many +O2 optimized SPECint95 binaries running under Dynamo is comparable to the performance of their +O4 optimized version running without Dynamo.

## Introduction

Recent trends in software technology have increased the obstacles to effective static compiler optimization. The use of object-oriented programming languages has resulted in a greater degree of delayed binding in modern software, thereby reducing the size of the static scope available to a traditional compiler. Shrink-wrapped software is increasingly being shipped as a collection of DLLs (dynamically linked libraries) rather than a monolithic binary. DLLs offer the advantage of being able to ship software patches to the customer after the product sale has been made, in addition to easing the integration of third-party middleware into the product. Unfortunately, it also makes traditional compiler optimization, which operates on the statically bound scope of the program, more difficult.

Compiler optimization as a performance delivery mechanism is also complicated by the prevalence of legacy binaries. Recompilation is clearly an impractical means for improving the performance of the huge volumes of legacy software. Finally, there is the emerging Internet and mobile communications marketplace. In a networked device where code can be downloaded and linked in on the fly, the role of traditional performance delivery mechanisms like static compiler optimization is unclear.

Thus, the effectiveness of the conventional model of performance delivery, where the task is divided only between the microprocessor hardware and the static compiler software, is limited in the emerging computing environment.

In this paper we propose a radically different performance delivery mechanism, one that is applied at the time the native bits are *executed* on the microprocessor, rather than when the native bits are *created* as in a compiler. We refer to this as *dynamic optimization*, and in principle, it is similar to what a superscalar reorder buffer does in a modern microprocessor: the *runtime optimization* of a *native* instruction stream. In contrast to a hardware dynamic optimizer however, our scheme works entirely in software, and yet retains the "transparent" nature of hardware dynamic optimization.

We have implemented a realistic prototype called Dynamo that demonstrates the feasibility of this idea, and represents the culmination of three years of research conducted at Hewlett-Packard Labs [Bala and Freudenberger 1996; Bala et al 1999]. Dynamo is a transparent dynamic optimizer that performs its optimization at runtime on a native program binary. Dynamo does not depend on any special compiler, operating system or hardware support, and it does not rely on any annotations in the input program binary. It uses interpretation as a means of observing program behavior without having to instrument it. No information is written out during or after the program's execution for consultation during a later run. Instead, profiling information is consumed on the fly to dynamically select hot instruction traces from the program. The hot traces are optimized using a low overhead optimizer and placed in a software code cache. Subsequent occurrences of these traces cause the cached version to be executed, giving a performance boost. The loaded binary image of the program code is untouched by Dynamo, allowing it to even handle programs that modify their own text image. The input program may also arm signal handlers. The operation of Dynamo is thus essentially transparent to the end-user.

Dynamic optimization is not intended to replace static compiler optimization, rather the two are for the most part complementary. The static compiler optimizer performs its optimizations based on the static scope of the program, whereas the dynamic optimizer operates based on the dynamic scope of the program, after all runtime objects have been bound. In contrast to a just-in-time (JIT) compiler, there is no translation component involved: the input is a native instruction stream. The dynamic optimizer can be viewed as performing a last minute optimization of the most frequently executed instruction sequences in the running program. This allows it to perform optimizations across dynamically linked procedure call boundaries and virtual function calls, which would be difficult to do in a static compiler. It can also perform runtime path-specific optimizations without having to do extensive code duplication to separate out the path from the surrounding control flow. Dynamic optimization opportunities exist even in programs compiled at the highest static optimization levels.

This paper presents an overview of the Dynamo system and highlights some of the technology innovations. A more comprehensive treatment can be found in [Bala et al. 1999].

The Dynamo prototype is implemented for the PA-RISC platform (a PA-8000 based workstation running HPUX 10.20). It is written in a combination of C and assembler, and the entire code footprint is 265 Kbytes. Although the implementation takes advantage of

some features of the PA instruction set architecture to improve the efficiency of certain functions, it does not rely on them for operation.

In this paper we present data for several integer benchmarks running under Dynamo, and compare their performance to the identical binary running directly on the processor. We only chose integer benchmarks because their inherent control flow complexity makes their optimization more challenging than relatively regular floating point codes. We show selected SpecInt95 benchmarks running the reference inputs, and a C++ code called *deltablue* which is an incremental constraint solver configured to solve 350,000 constraints [Sannella et al. 1993]. The SpecInt95 benchmark *gcc* is not included in the benchmark set shown here because the SPEC version is actually a script that repeatedly invokes gcc on a number of different inputs. A program must run at least 2 mins on the PA-8000 for Dynamo to be able to provide any performance benefit; although the *gcc* benchmark as a whole is long running, Dynamo only sees it as a collection of separate runs of gcc, many of which are too short to optimize profitably. The behavior of a custom monolithic version of the *gcc* benchmark running under Dynamo closely matches that of *go* for all of the data shown in this paper. All programs with the exception of *deltablue* were compiled with the product HP C compiler, at the default +O2 optimization level (equivalent to –O). *deltablue* was compiled with g++ because it produced a faster running binary than the HP C++ compiler. All performance data are based on actual wall-clock times when running the binary under Dynamo and are compared to the identical binary running directly on the PA-8000.

## How Dynamo Works

Figure 1 shows a high level outline of how Dynamo works. Dynamo starts out by interpreting the input program's instructions (box A). The input program must be a user-mode native executable (supervisor-mode programs can create complications that our current prototype cannot yet handle reliably). It can invoke dynamically linked libraries (shared libraries) and system calls. It can also arm signal handlers. The "interpreter" here is a *native* instruction interpreter, implemented in software. Interpretation is primarily used as a means for observing program behavior without having to instrument the program binary. As a side effect of interpretation, Dynamo increments counters associated with certain program addresses that satisfy a "start-of-trace" condition (such as the interpretation of a backward taken branch, boxes C and D). Path ABCA and the counter-update path ABCDEA in the figure represent the operation of the interpreter in its "normal mode". If a counter value exceeds a hot threshold (box E), the interpreter toggles state and goes into its "trace generation mode" (box G). When interpreting in this mode, the native instruction(s) corresponding to each interpreted instruction is emitted into a buffer. Path GHG represents the "trace generation" interpretive loop. When an "end-of-trace" condition is reached, the contents of the buffer are optimized by a fast, lightweight optimizer (box I). The optimizer makes the trace instructions into an executable unit called a *fragment*, performs some lightweight optimizations on the fragment, and then hands it to a linker (box J). The linker emits the optimized fragment code into a software code cache called the *fragment cache*, and links it to other fragments already in the cache. The fragment is tagged with the program address corresponding to the first instruction in the sequence.

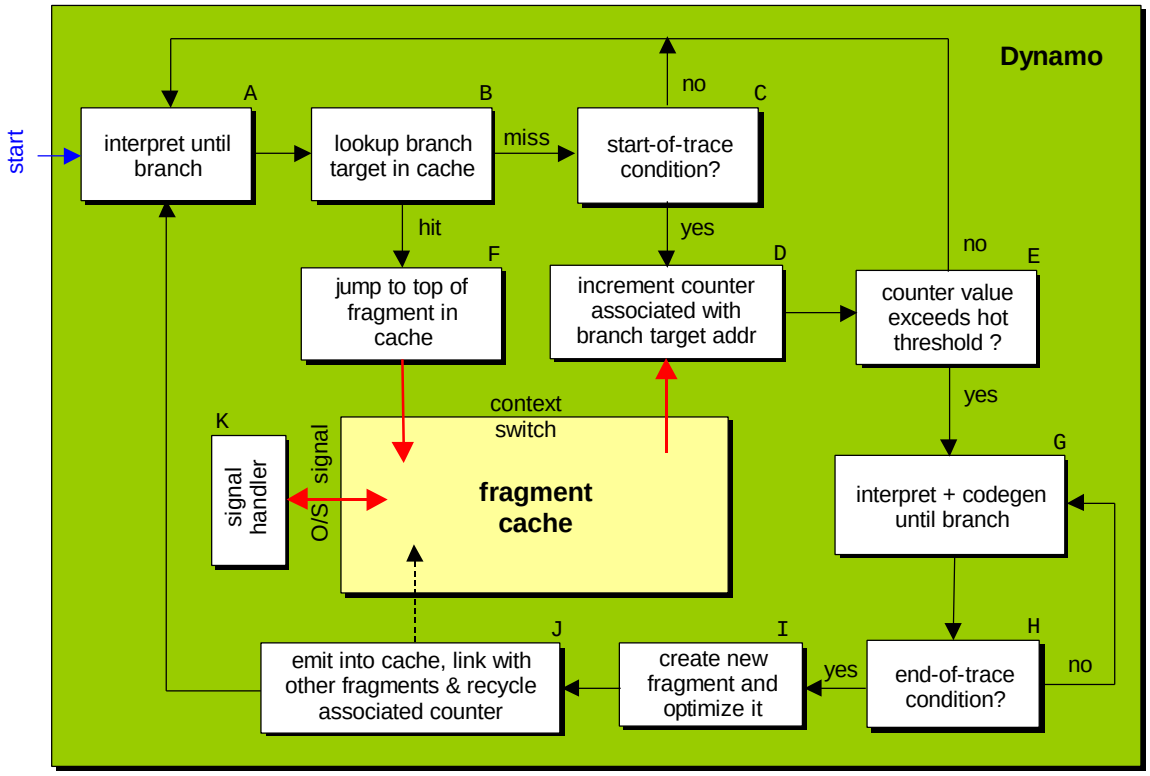


**Figure 1:** How Dynamo works

Every time the interpreter, when interpreting in its normal mode, interprets a branch instruction, a fragment cache lookup is done to check if a fragment exists whose tag is equal to the branch target PC (box B). Only program addresses that satisfy the "start-of-trace" condition can be entry points of control within the fragment cache. Control never jumps into the interior of a fragment. Upon a fragment cache hit, control jumps to the top of the fragment, causing that fragment code (and others that may be linked to it) to execute directly on the underlying processor (box F). At this point, the execution of Dynamo is effectively suspended, and the program's instructions (actually their optimized version in the software fragment cache) are executing directly on the processor. The optimized fragments from the fragment cache gradually start populating the underlying processor's instruction cache, giving a performance boost.

Eventually, a branch exits the fragment cache address space, causing control to trap to Dynamo. The execution of the dynamically optimized program within the fragment cache is now suspended, and Dynamo code is executing on the underlying processor. A counter associated with the exit branch target is incremented (box D), and if this counter exceeds the hot threshold, the interpreter is invoked in its trace generation mode, causing it to produce a fresh trace. Otherwise, the interpreter is invoked in its normal mode of operation, completing the Dynamo execution loop. The rationale here is that if an exit from a hot trace itself gets hot, a new trace should be formed starting at that exit branch target.

Gradually, as more and more hot traces materialize in the fragment cache, an increasingly greater time is spent executing within the fragment cache, and a correspondingly smaller proportion of time is spent in Dynamo.

## Dynamo overhead

Clearly, for such a strategy to provide a performance boost, a significantly greater amount of time must be spent executing within the fragment cache than executing within Dynamo. This means Dynamo can afford to interpret only cold parts of the program code. The hot parts of the code have to be quickly materialized into the fragment cache so that they can execute directly on the underlying processor without incurring any overhead.

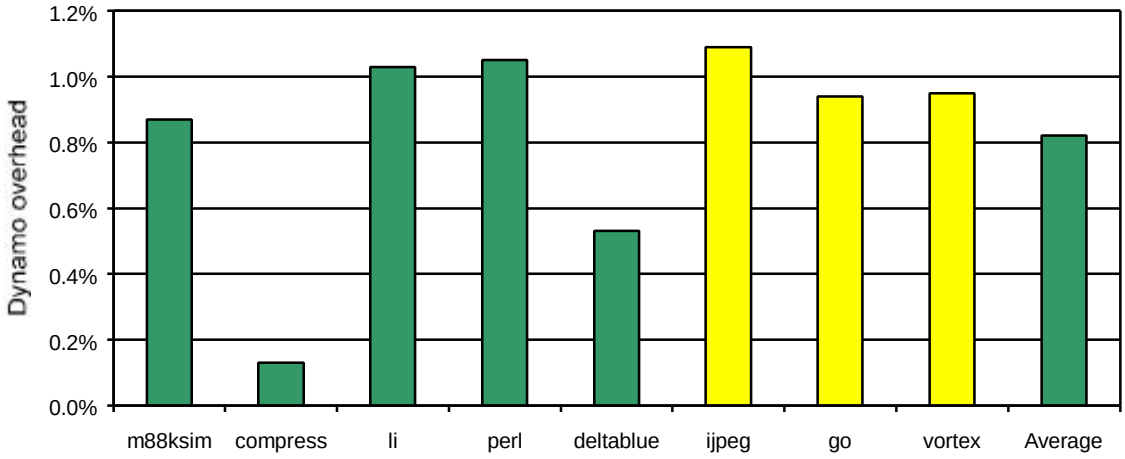


**Figure 2:** Dynamo overhead as a percentage of total execution time (lightly shaded bars indicate that Dynamo bailed out)

The choice of the threshold at which a start-of-trace address is considered hot (box E) is therefore critical in controlling the overall Dynamo overhead. Intuitively, the slower the interpreter, the quicker the program's working set must be captured within the fragment cache, since the cached code executes directly on the underlying processor without incurring any overhead. However, the smaller the hot threshold value, the more speculative the trace selection scheme becomes, because Dynamo does not get to observe the program behavior long enough to make a more judicious choice. On some programs, decreasing the hot threshold may have the undesirable side effect of increasing the number of traces that become hot. This has a ripple effect on the fragment cache design, which, in order to compensate for a potentially high rate of fragment generation, has to be sized large enough to keep the cache replacement overhead low. Furthermore, an increase in the number of hot traces may also increase the optimization overhead incurred by Dynamo. All of these tradeoffs have to be kept in balance in choosing the hot threshold value. In our prototype, a threshold of 50 was found to be the sweet spot. Thus, a start-of-trace has to be interpreted at least 50 times before a trace starting at that address is selected for optimization and code generation into the fragment cache.

Whatever the overhead, it must be offset by a sufficient performance boost just to break even with the performance of the input native program running directly on the processor. The hot traces generated into the fragment cache must therefore be executed long enough to more than offset the overhead of creating them in order for Dynamo to provide any performance benefit. For very short running programs (running time < 2 mins on a PA-8000), it is unlikely that this will happen. The benchmark *ijpeg* is an example of such a case. However, for longer running programs where the 90/10 rule

applies (i.e., approximately 90% of execution time is spent in 10% of the code), Dynamo can be highly effective. Figure 2 shows the overall Dynamo overhead as a percentage of the total program execution time under Dynamo. The average overhead for these benchmarks is less than 1%. Thus, nearly 99% of the execution time is typically spent executing in the fragment cache.

Some programs however, are not very well behaved, in the sense that they do not spend a significant amount of time in a stable working set. By the time Dynamo has captured the current hot traces in the fragment cache, the program's working set changes. The benchmarks *go* and *vortex* fall into this category. In order to recognize this situation, Dynamo constantly monitors the rate of new fragment creation, and if this rate exceeds a tolerable threshold for a prolonged interval, Dynamo assumes that the program is ill behaved. At that point, Dynamo *bails out*, and lets the input program run directly on the underlying processor. This is possible because the input to Dynamo is itself a native program binary. Figure 3 illustrates the effect of bailing out on the benchmark *go*. The relatively well-behaved program *li* is also shown for comparison. Bailing out allows Dynamo to deliver close to break-even performance on ill-behaved programs such as *go*. However, a drawback of the bail-out strategy is that once Dynamo bails out, it never regains control over the program. This may cause Dynamo to give up too early on some programs.

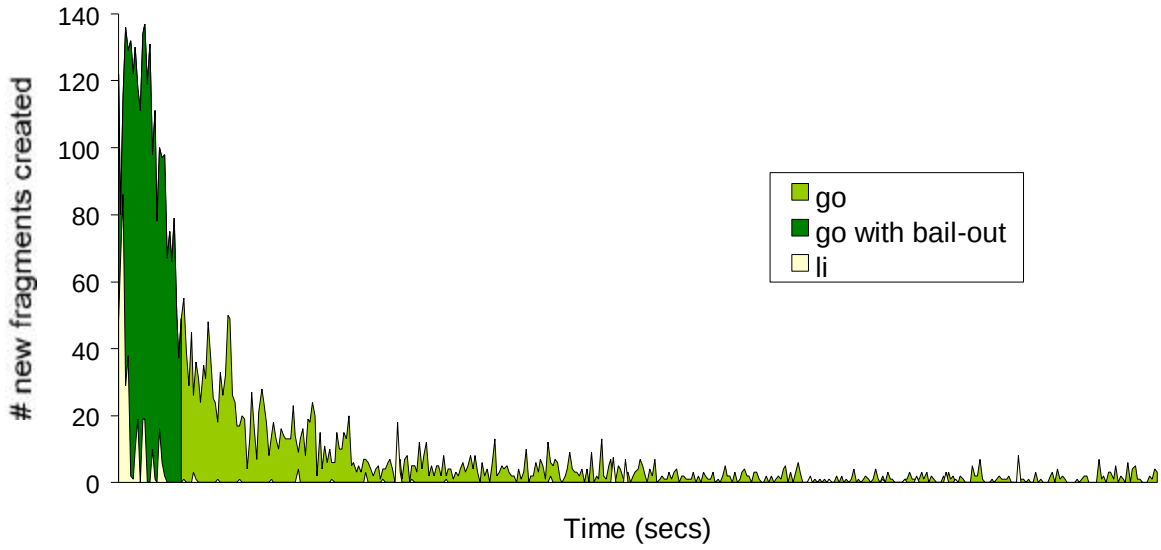


**Figure 3:** Effect of bailing out based on the fragment creation rate

## Starting up Dynamo

The entire execution of the input program is done under Dynamo control. The original application program instructions never execute at their original program text addresses. Program instructions are either interpreted by Dynamo or their optimized version is executed within the software fragment cache that is managed by Dynamo. The same is true of the instructions in any dynamically linked libraries that may be invoked by the program. Operating system calls on the other hand execute directly on the underlying processor. Dynamo does not attempt to follow a call into the operating system

code. Instead, it sometimes intercepts the operating system call and modifies the context of the call to ensure that control will return to Dynamo after the system call completes.

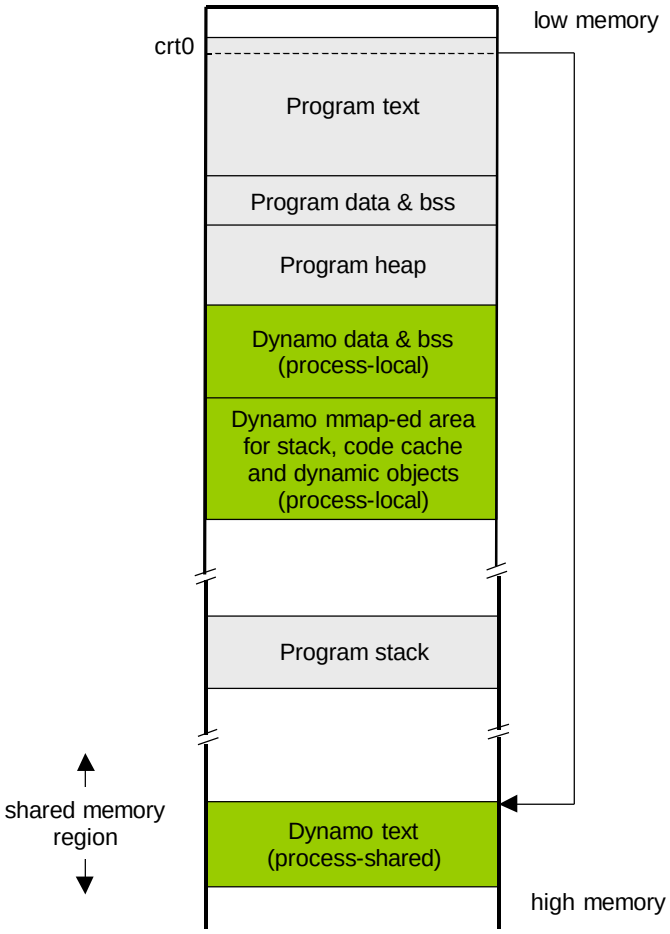


**Figure 4:** Memory layout of a program executing under Dynamo

There are several ways to set up Dynamo to gain control of the program after its binary image has been loaded in memory, before any of the program instructions have been executed. One approach is to use a special version of *crt0*, the execution start up prologue at the top of the program's binary image. Dynamo is compiled as a shared library that is pre-loaded into the shared memory segment of the virtual address space. Prior to jumping to the program entry point, *crt0* checks if the Dynamo shared library is currently mapped in memory. If it is, *crt0* will call Dynamo's entry point instead of the one defined by the _start symbol (the program's entry point), and the program will then execute under Dynamo control. Otherwise, it will jump to the program entry point, and the program will execute in the normal manner.

There are alternative approaches that do not require a custom version of crt0, that allow even legacy binaries to be run under Dynamo. Details can be found in [Bala et al. 1999].

The actual program's instructions are unaltered by this technique. The program is loaded at the identical address in memory as it would be if it ran directly on the processor. This is essential in order to meet our transparency goal. For example, certain

classes of program bugs that are dependent on the mapping of the program text segment in memory cannot be reproduced if its mapping is altered from the original.

The Dynamo shared library is unique with respect to other shared libraries. It does not use any part of the program's data segment when it executes. Instead, when control first enters the Dynamo shared library, an initialization routine *mmaps* a separate area of memory that is used to allocate internal data structures and a custom runtime stack. No recursive routines are used within Dynamo, and the longest call-chain within Dynamo is known a priori. This allows Dynamo to allocate a fixed amount of room for its runtime stack. Dynamo does not use a malloc-style heap; the separate *mmap*-ed area is used to allocate its dynamic objects, using a custom memory allocator that is part of Dynamo. Thus, Dynamo's operation does not interfere with the application program's runtime stack or heap area, another critical requirement for meeting our transparency goals. Figure 4 illustrates the typical memory layout of a program running under Dynamo.

## Context switching

At any given moment during program execution under Dynamo, control is either in the fragment cache or in the Dynamo code itself. Thus, two different machine contexts exist, one that corresponds to the execution of Dynamo, and the other to the execution of the dynamically generated code in the fragment cache. When control is in Dynamo, the underlying machine registers contain Dynamo's context, and the application program's context is kept in a context save area in the Dynamo data segment. When control is transferred to the fragment cache, the application program's context is loaded into the underlying machine registers, but Dynamo's context is discarded. Thus, Dynamo never "resumes" execution once control leaves it and enters the fragment cache.

The saving and restoring of the program's context is done by Dynamo and not by the operating system. This keeps the context switch fast. Dynamo only saves and restores a partial program context, keeping one register permanently in the context save area so that it is available for implementing the context switch to exit the fragment cache.

## Trace selection

Dynamo's granularity of hot spot selection is a dynamic instruction trace. Because the trace is selected dynamically during program execution, the trace instructions are often non-contiguous in the program memory. The trace can go through statically and dynamically linked procedures, indirect branches and virtual function calls.

Dynamo uses a novel approach for hot trace selection that is highly speculative in nature and involves minimal profiling. The essence of the idea is that when a program address gets hot, the sequence of addresses executed immediately following it is statistically likely to be a hot trace. Thus, instead of using expensive branch or path profiling techniques [Ball and Larus 1996], Dynamo only profiles a small set of special program addresses, which are considered good candidates to start a trace.

These special program addresses must meet one of the following start-of-trace conditions: (a) it is the target of a backward taken branch (likely to be a loop head), or (b) it is the target of an exit branch from a previously selected hot trace in the fragment cache. The counter is incremented only when the start-of-trace address is interpreted (and control is in Dynamo). No counter update or any other kind of profiling is done when control is in the fragment cache.

When the counter associated with a start-of-trace address exceeds a threshold (50 in our prototype), the interpreter toggles to the code generation mode, emitting each interpreted native instruction into a buffer. This allows the dynamic sequence of instructions executed immediately following the hot start-of-trace instruction to be selected as a hot trace. An advantage of this approach is that the trace can be grown even past an indirect branch or dynamically linked library call. The interpreter continues to emit the hot trace into a buffer until it reaches one of the following end-of-trace conditions: (a) a backward taken branch is interpreted, or (b) the buffer is full. When an end-of-trace condition is reached, the counter associated with the start-of-trace address is reset and recycled for profiling future start-of-trace addresses, and the interpreter toggles back to its normal state of operation.

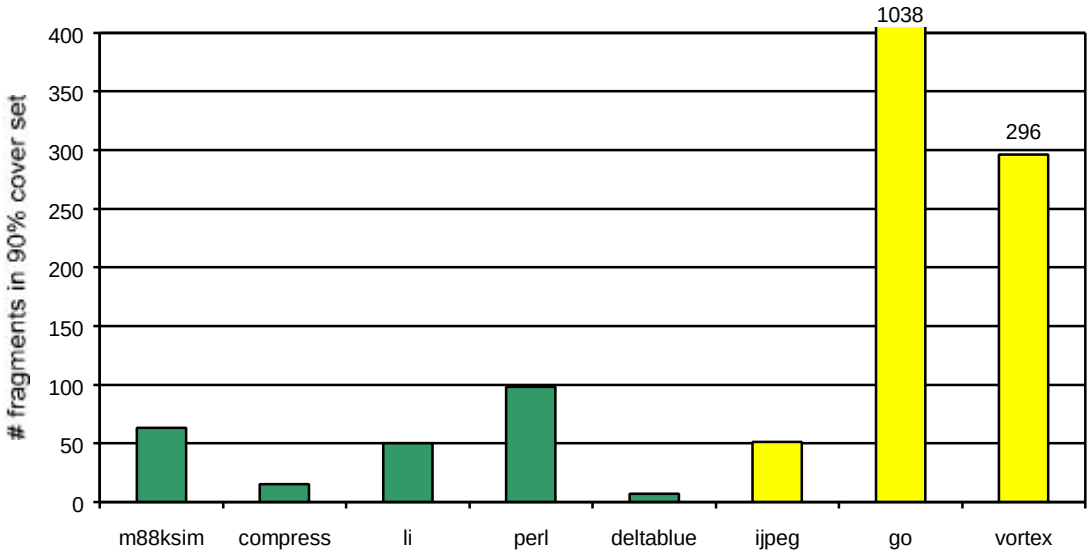


**Figure 5:** Smallest number of fragments accounting for 90% of total execution time (lightly shaded bars are programs on which Dynamo would normally bail out, but this data was collected with bail-out was disabled)

The profile information associated with an address is thus consumed and discarded as soon the address gets hot (box J in Figure 1). This *online* profiling strategy is in contrast to conventional *offline* profiling techniques that accumulate profile information during a test run and feed the profile information back to a compiler or analysis tool. Furthermore, in the case of conventional techniques, once a counter is associated with a profiled address, its storage is locked to that address until the profiling run terminates. The ability to recycle the counter storage once the counter value gets hot allows our scheme to be more economical in its use of counter memory.

The speculative trace selection scheme outlined above selects hot dynamic traces without using any kind of path profiling. Yet, it is very effective as illustrated by the data in Figure 5. The data shows the smallest number of fragments in the fragment cache that account for 90% of the total program execution time under Dynamo (i.e., the top *n* fragments that account for 90% of the total execution time). On the programs where Dynamo does not bail out, this 90% cover set typically contains less than 100 fragments. *go* and *vortex* have a much larger number of fragments in the 90% cover set, but Dynamo

bails out to direct native execution in these cases, long before this many fragments are generated into the fragment cache. In our extensive experiments with alternative region selection schemes, we were unable to find one that matched the performance/overhead ratio of our speculative trace selection scheme [Bala et al. 1999].

## Fragment creation and optimization

The instructions comprising a hot trace are often non-contiguous in memory, and thus may not have been part of the static program scope that was available to the compiler that created the program binary. A dynamic trace is therefore likely to expose new optimization opportunities, even in a statically optimized program binary.

The first task of the optimizer (box `I` in Figure 1) is to fix up taken branches on the trace so that the fall-through direction remains on the trace and the taken direction exits the trace. This allows the trace instructions to be placed contiguously in the fragment cache memory as an executable *fragment*. Sometimes a branch may be redundant on the trace and can be removed altogether. An example is the occurrence of a procedure call and its corresponding return on the same trace. Both the call and the return are redundant branches. Any side effects of the branch instruction (such as setting a link register) are preserved if the branch is removed. When creating an executable fragment from the trace, the optimizer also inserts special blocks called linker stubs that serve as the targets of branches that exit the fragment. Every exit branch from the fragment targets a unique linker stub. The purpose of the linker stub is to trap control back to Dynamo. It does so by jumping to a context switch routine, which saves the program's context into Dynamo's context save area and then transfers control to Dynamo. The linker stub also communicates the actual target of the exit branch to Dynamo, so that Dynamo can resume interpretation starting at that target address. Figure 6 (a) shows a trace in the input program, and (b) shows its corresponding executable fragment in the fragment cache.

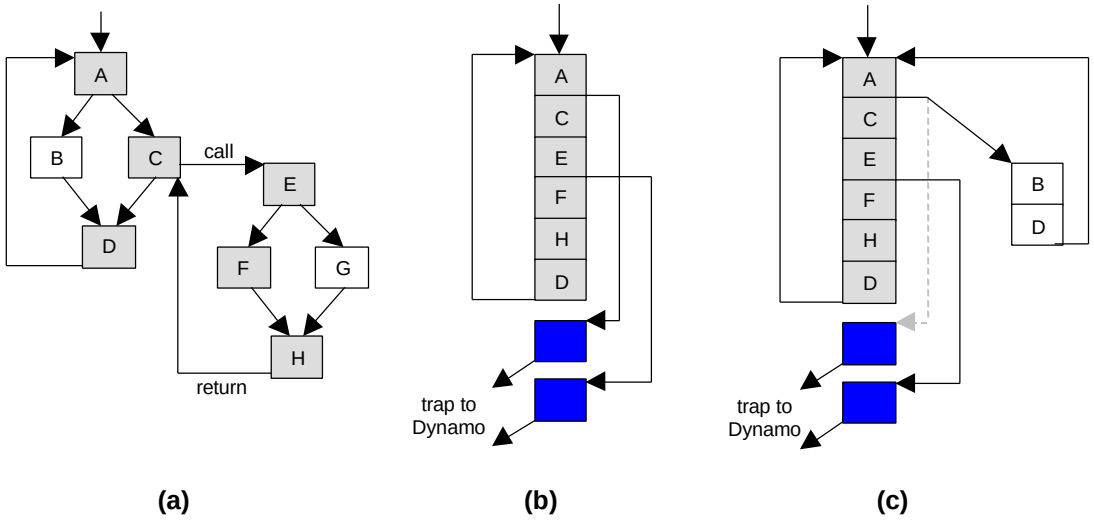


**Figure 6:** (a) Control flow snippet in input program, with hot trace blocks shown lightly shaded, (b) Fragment corresponding to the hot trace as laid out in the fragment cache, with linker stubs shown darkly shaded, and (c) Illustration of branch linking between fragments

Indirect branches (i.e., branches that use the value of a register as the target) on the trace are handled in a special way. When a trace contains an indirect branch, the instruction immediately following the branch is the indirect branch target that was

encountered at the time the trace was interpreted. Thus, the trace automatically contains one inlined indirect branch target. The indirect branch is replaced by a conditional branch that tests if the value in its target address register is the same as the address of the inlined target. If the test fails, the branch exits the trace to a special fragment that is permanently kept in the fragment cache. This special fragment is a hand-coded sequence that does a tag lookup of the fragment cache. If a fragment is found for the actual target, control jumps to it, otherwise it exits the fragment cache to Dynamo, which will then invoke the interpreter.

A fragment is a single-entry multi-exit executable sequence. Control can only enter a fragment at the top. There are no internal control join points in a fragment. This simplicity of control flow allows a very fast, non-iterative optimizer design that is highly effective. Dynamo does not use any form of "staged optimization". Rather, a number of path-specific optimization passes are performed on each trace. The optimizations are divided into two classes: *conservative* and *aggressive*. Optimizations are termed conservative if they do not lead to modified program memory or register contents. Elimination of redundant branches (that result as a side effect of trace selection), copy propagation, constant propagation and strength reduction fall into this class. Aggressive optimizations include redundant load and store removal, dead code elimination and loop transformations. Aggressive optimizations can be unsafe in certain situations. For example, a redundant load may actually be a volatile memory accesses, and its removal could result in an incorrect program. Aggressive optimizations are also difficult to undo. For instance, the program might arm a signal with a handler that examines or even modifies the program's machine context at the instant of the signal. If this signal arrives when control is in the middle of a fragment in the fragment cache, Dynamo has to reconstruct the original signal context as if the program were running directly on the processor. This may be difficult to do if aggressive optimizations were applied on that fragment. Although Dynamo is capable of starting out in the aggressive mode and switching to the conservative mode (accompanied by a fragment cache flush) when it encounters any suspicious code, our prototype does not exercise this. This is because source program information such as volatile memory access is not preserved in a PA-RISC binary. The decision as to whether to perform conservative or aggressive optimizations is therefore made when Dynamo is first started.

Figure 7 shows actual wall-clock speedups that Dynamo achieves over +O2 optimized[1] native program binaries running without Dynamo. On average Dynamo achieves a speedup of 11.4%. In comparison, the average speedup over the same binaries achieved by +O4 static compiler optimization[2] is only 7.3%. Thus, the performance of the above +O2 binaries running under Dynamo is 4.1% faster on average than their +O4 compiled versions running without Dynamo. Importantly, Dynamo achieves this speedup in a transparent fashion without resorting to any kind of profile-based feedback mechanism.

[1] +O2 is equivalent to the –O default optimization level in the HP C compiler.

[2] +O4 is the highest optimization level that does not require a profile feedback run in the HP C compiler. Several global optimizations, including procedure inlining, are performed at this level. Compile time at +O4 is significantly slower than at +O2.

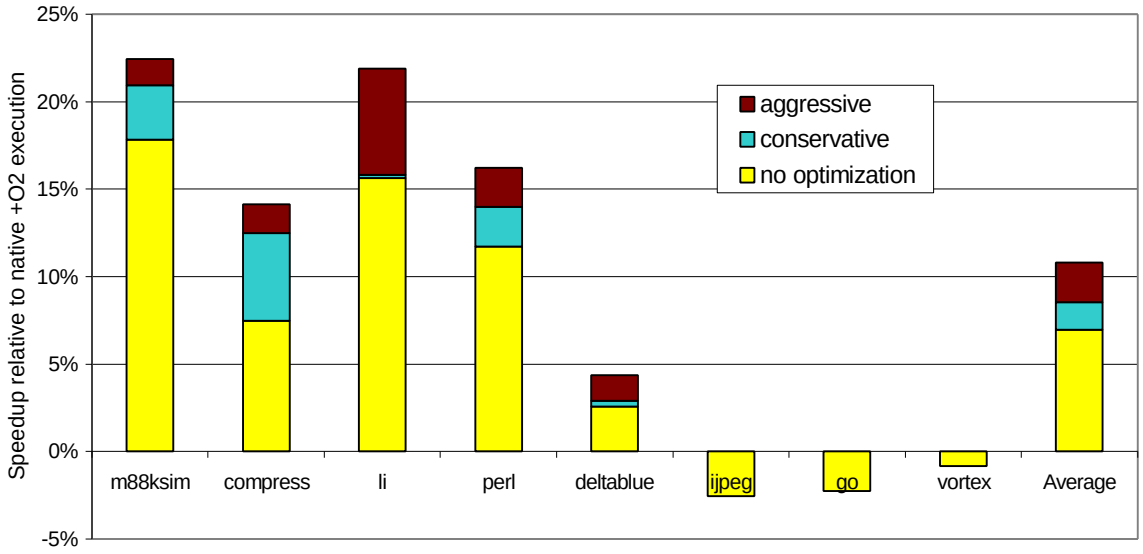


**Figure 7:** Dynamo performance relative to direct execution of the input native program

As the figure indicates, a significant portion of the performance gains come from trace selection itself. This is primarily due to the compact layout of program hot spots in the fragment cache memory, which has desirable side effects such as improved instruction cache utilization and lower virtual memory pressure [Bala et al. 1999]. Trace optimization accounts for approximately 5% of the total gains. The time spent in performing the optimizations is a negligible fraction of the overall Dynamo overhead (on average < 0.5%). Even so, this overhead is invested very efficiently, since optimization is only attempted on dynamic hot code.

## Branch linking

When a new fragment is inserted into the fragment cache, its exit branches are linked to jump directly to other fragments in the cache if their tags match the original exit branch addresses (box J in Figure 1). This involves a simple modification to the branch instruction (Figure 6 (c)). Branch linking significantly lowers the number of fragment cache exits, and hence the context switch overhead incurred by Dynamo. Linking is critical for Dynamo to deliver performance. If linking is disabled, Dynamo performance can slow down significantly (a factor of 29 average slowdown in our current prototype).

## Fragment cache management

For the performance data shown in Figure 7 Dynamo's fragment cache was sized at 500 Kbytes. Dynamo cannot afford to do fancy management of the fragment cache storage, because of the overheads it would incur. One approach is to simply size the cache large enough to hold the entire working set of a typical input program. This is the approach followed by most contemporary dynamic translator implementations. Dynamo on the other hand uses a pre-emptive flushing scheme to periodically remove the entire contents of its fragment cache. The flushing is only triggered when Dynamo recognizes a sharp increase in the fragment creation rate. The rationale here is that a sharp rise in new

fragment creation is indicative of a change in the working set of the program that is currently in the fragment cache. Since a new working set is being built, a fragment cache flush is unlikely to displace useful fragments[3]. Use of this pre-emptive flushing technique permits a 30% reduction in the fragment cache footprint without any performance loss (i.e., the same performance of Figure 7 is achieved with a 350 Kbyte fragment cache).

Besides allowing a smaller size fragment cache to achieve the same performance as a larger one, the pre-emptive flushing mechanism has other interesting side effects. All fragment related data structures maintained for internal bookkeeping by Dynamo are tied to the flush, causing these memory pools to be reset as a side effect of a pre-emptive flush. A pre-emptive flush thus serves as a cheap garbage collecting mechanism to free dynamic objects associated with fragments that are likely to have dropped out of the current working set. If some fragments belonging to the new working set are inadvertently flushed as a result, they will simply be regenerated by Dynamo when those program addresses are encountered during execution.

## System Calls and Signal Handling

Dynamo examines all system calls in the program code. Usually, this is necessary to identify situations where the kernel might directly invoke a program routine, undercutting Dynamo and causing it to lose control over the program. An example is system calls that install signal handlers. Dynamo intercepts all signal handler installation calls (such as *sigaction*), and substitutes its own handler in place of the program's handler. The kernel will transfer control to Dynamo when the signal arrives (box K in Figure 1), and the program's handler code is then executed under Dynamo control.

Sometimes however, the situation can get quite complicated, and Dynamo has to make a decision on whether it can continue to maintain control over the program without compromising performance. An example of this situation is when the program code contains a call to *fork* or *exec*. In principle, it is possible for Dynamo to inject itself into the executable image of the child process, similar to the way the current process was set up to run under Dynamo. Rather than deal with the additional complication this creates however, our prototype currently bails out to direct program execution if a *fork* or *exec* is encountered in its code.

## Related Work

A lot of work has been done on dynamic translation as a technique for non-native system emulation [May 1987; Insignia 1991; Cmelik and Keppel 1993; Stears 1994; Bedichek 1995; Witchel and Rosenblum 1996; Hohensee et al. 1996; Herold 1998]. The basic idea is simple: interpretation is too expensive a way to emulate a long running program on a host machine. Caching the native code translations of frequently interpreted regions can lower the overhead. If sufficient time is spent executing the cached translations to offset the overhead of interpretation and translation, the caching strategy will generally beat pure interpretation in performance. Some implementations even bypass the interpreter completely and translate to native code prior to executing every unit [Conte and Sathaye 1995; Cramer et al. 1997; Ebcioglu and Altman 1997; Griswold 1998]. Others, like Dynamo, employ a blend of interpretation and native execution

[3] If the fragment creation rate continues to remain high, Dynamo will bail out as illustrated in Figure 3.

[Ebcioglu et al. 1999]. More recently, hybrid dynamic translation systems are emerging that use a combination of software and custom hardware [Kelly et al. 1998].

Dynamic *optimization* is different from dynamic translation: the input is a *native* program binary, so there is no "translation" step involved. While the goal of a dynamic translator is to beat the performance of pure interpretation, the goal of a dynamic optimizer is to beat the performance of the input program executing directly on the host machine.

Dynamic optimization is also different from dynamic *compilation* as generally defined in the literature. In dynamic compilation, the programmer either explicitly requests that a specific part of the program be compiled at runtime via program annotations [Auslander et al. 1996; Consel and Noel 1996; Leone and Lee 1996; Leone and Dybvig 1997 [4]] or uses a language that allows efficient runtime compilation [Holzle 1994; Engler 1996; Poletta et al. 1997]. The code generated at compile-time contains stubs to transfer control at runtime to a compiler, which then dynamically generates the native executable code. In contrast, we view dynamic *optimization* as a much more transparent process. No special programming language or compiler annotations should be necessary to trigger its activation. It should work on already compiled native code without the need for any additional processing, including instrumentation of the binary. In the case of dynamic compilation, the runtime compiler is a necessary step for the execution of the program, whereas in the case of dynamic optimization, it is not. At any moment, the dynamic optimizer has the option of "bailing out" and letting the input program execute directly on the underlying processor.

There are several implementations of *offline binary translators* that also perform native code optimization [Sites et al. 1992; Chernoff et al. 1998]. These generate profile data during the initial run via emulation, and perform background translation together with optimization of hot spots based on the profile data. Again, this strategy is not transparent in the sense that the dynamic optimizer is. The benefit of the optimization is only available during subsequent runs of the program. In the case of a dynamic optimizer, on the other hand, all profile data that is generated is consumed in the very same run, and no data is written out for use offline or during a later run.

Hardware implementations of dynamic optimizers are now commonplace in modern microprocessors [Kumar 1996; Papworth 1996; Keller 1996]. The optimization unit is a fixed size instruction window, with the optimization logic operating on the critical execution path. The Trace Cache is another hardware alternative that can be extended to do superscalar-like optimization off the critical path [Peleg and Weiser 1994; Rotenberg et al. 1996; Friendly et al. 1998]. But we are unaware of any attempts to implement a transparent dynamic optimizer entirely in software. A software dynamic optimizer has the advantage that it can be patched similar to the BIOS on a PC. It is also not bound to a particular hardware implementation, and can also be custom-tuned to specific application domains easily.

[4] The Indiana project is also called "Dynamo". We regret the name collision. When our project was started in the winter of 1995, we were unaware of the other Dynamo project. We have since used the Dynamo name in numerous internal documents, memos, presentations and patent filings, making the task of renaming our project at this stage somewhat more difficult than one might imagine.

## Conclusion

Dynamo offers a novel software performance delivery technology: performance optimization is done at the time the bits are executed on the client machine, rather than at the time the bits are created at compile time. The advantage of this approach is that one need not rely on the software vendor to enable the optimization level. Our prototype has already demonstrated that significant performance wins can be achieved, even on integer benchmarks compiled with static optimization. In addition, delayed binding and dynamic linking actually work in our favor, because these do not present obstacles to a dynamic optimizer that operates after such objects have been bound at runtime.

The Dynamo prototype has shown that it is possible to get significant speedups through dynamic optimization, even from statically optimized native binaries. It has also demonstrated the feasibility of engineering a realistic transparent dynamic optimizer, in spite of the severe constraints of operating at runtime.

## References


Auslander, J., Philipose, M., Chambers, C., Eggers, S.J., and Bershad, B.N. 1996. Fast, effective dynamic compilation. In *Proceedings of the SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI'96*).

Bala, V., Duesterwald, E., and Banerjia, S. 1999. Transparent Dynamic Optimization: The Design and Implementation of Dynamo. *Hewlett Packard Laboratories technical report HPL-1999-78, June 1999.* Available at http://www.hpl.hp.com/techreports/1999/HPL-1999-78.html.

Bala, V. and Freudenberger, S. 1996. Dynamic optimization: the Dynamo project at HP Labs Cambridge (project proposal). *Hewlett Packard Laboratories internal memo, Feb 1996.*

Ball, T. and Larus, J.R. 1996. Efficient path profiling. In *Proceedings of the 29th Annual International Symposium on Microarchitecture (MICRO-29)*, Paris. 46-57.

Bedichek, R. 1995. Talisman: fast and accurate multicomputer simulation. In *Proceeding 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*.

Chernoff, A., Herdeg, M., Hookway, R., Reeve, C., Rubin, N., Tye, T., Yadavalli, B., and Yates, J. 1998. FX!32: a profile-directed binary translator. *IEEE Micro, Vol 18, No. 2, March/April 1998.*

Cmelik, R.F. and Keppel, D. 1993. Shade: a fast instruction set simulator for execution profiling. *Technical Report UWCSE-93-06-06, Dept. Comp. Science and Engineering, Univ. Washington.*

Consel, C. and Noel, F. 1996. A general approach for run-time specialization and its application to C. In *Proceedings of the 23th Annual Symposium on Principles of Programming Languages.* 145-156.

Conte, T.M. and Sathaye, S.W. 1995. A technique for object code compatibility in VLIW architectures. In *Proceedings of the 28th Annual International Symposium on Microarchitecture (MICRO-28).* 208-218.

Cramer, T., Friedman, R., Miller, T., Seberger, D., Wilson, R., and Wolczko, M. 1997. Compiling Java Just In Time. *IEEE Micro*, May/Jun 1997.

Ebcioglu K. and Altman, E.R. 1997. DAISY: Dynamic compilation for 100% architectural compatibility. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*. 26-37.

Ebcioglu K., Altman, E.R., Sathaye, S, and Geschwind, M. 1999. Execution-based scheduling for VLIW architectures. To appear in *Euro-Par, August '99, Toulouse, France.*

Engler, D.R. 1996. VCODE: a retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI'96*).

Friendly, D.H., Patel, S.J., and Patt., Y.N. 1998. Putting the fill unit to work: dynamnic optimizations for trace cache microprocessors. In *Proceedings of the 31st Annual Internation Symposium on Microarchitecture (MICRO-31),* Dallas. 173-181.

Griswold, D. 1998. The Java HotSpot virtual machine architecture. *Sun Microsystems, Mar. 1998. Available from http://java.sun.com/products/hotspot/whitepaper.html.*

Herold, S.A. 1998. Using complete machine simulation to understand computer system behavior. *Ph.D. thesis, Dept. Computer Science, Stanford University*.

Hohensee, P., Myszewski, M., and Reese, D. 1996. Wabi CPU emulation. In *Proceedings Hot Chips VIII*.

Holzle, U. 1994. Adaptive optimization for SELF: reconciling high performance with exploratory programming. *PhD thesis, Computer Science Dept., Stanford University, available as Technical Report STAN-CS-TR-94-1520. Also available as a Sun Microsystems Lab technical report.*

Insignia 1991. SoftWindows for Macintosh and UNIX. http://www.insignia.com.

Keller, J. 1996. The 21264: a superscalar Alpha processor with out-of-order execution. Presented at the *9th Ann. Microprocessor Forum*, San Jose, CA.

Kelly, E.K., Cmelik, R.F., and Wing, M.J. 1998. Memory controller for a microprocessor for detecting a failure of speculation on the physical nature of a component being addressed. *U.S. Patent 5,832,205, Nov. 1998.*

Kumar, A. 1996. The HP PA-8000 RISC CPU: a high performance out-of-order processor. In *Proceedings Hot Chips VIII*, Palo Alto, CA.

Leone, M. and Dybvig, R.K. 1997. Dynamo: a staged compiler architecture for dynamic program optimization. *Technical Report #490, Dept. Computer Science, Indiana University.*

Leone, M. and Lee, P. 1996. Optimizing ML with run-time code generation. In *Proceedings of the SIGPLAN'96 Conference on Programming Language Design and Implementation.* 137-148.

May, C. 1987. Mimic: a fast System/370 simulator. *ACM SIGPLAN 1987 Symposium on Interpreters and Interpretive techniques.*

Papworth, D. 1996. Tuning the Pentium Pro microarchitecture. *IEEE Micro*, (Apr.). 8-15.

Peleg, A. and Weiser, U. 1994. Dynamic flow instruction cache memory organized around trace segments independent of virtual address line. U.S. patent 5,381,533.

Poletta, M., Engler, D.R., and Kaashoek, M.F. 1997. tcc: a system for fast flexible, and high-level dynamic code generation. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*. 109-121.

Rotenberg, E., Bennett, S., and Smith, J.E. 1996. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th Annual International Symposium on Microarchitecture (MICRO-29),* Paris. 24-35.

Sannella, M., Maloney, J., Freeman-Benson, B., and Borning, A. 1993. Multi-way versus one-way constraints in user interfaces: experiences with the DeltaBlue algorithm. *Software – Practice and Experience 23,* 5 (May). 529-566.

Sites, R.L., Chernoff, A., Kirk, M.B., Marks, M.P., and Robinson, S.G. Binary Translation. *Digital Technical Journal, Vol 4, No. 4, Special Issue, 1992.*

Stears, P. 1994. Emulating the x86 and DOS/Windows in RISC environments. In *Proceedings Microprocessor Forum*, San Jose, CA.

Witchel, E. and Rosenblum R. 1996. Embra: fast and flexible machine simulation. In *Proceedings of the SIGMETRICS '96 Conference on Measurement and Modeling of Computer Systems.* 68-78.